A High-Level Approach to Computer Document Formatting

Brian K. Reid

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

1. Introduction

The very best document-formatting system is a good secretary. He can be given scrawled handwritten text in no particular format, and without further instruction produce a flawless finished document. Nevertheless, we believe that document formatting should be done by computers, because so much of it is the tedium that computers handle so well. Existing computer document formatting programs have met with some success; indeed, most computer systems offer some sort of text formatting capability. These programs are often difficult to use, and are almost invariably tied to a particular kind of printing device.

The document-formatting language *Scribe* was designed to provide a simple, portable language in which document formatting could be specified; the Scribe compiler was written to process that language into finished documents. In following sections we describe the design goals, the implementation, and report on experience with the completed system.

2. Prior Work

Typical text processing programs, such as PUB, TεX, TROFF, Script, DPS, Texture, Textform, Text360, etc. [1, 4, 7, 10, 11], require the user to provide them with an input stream that consists of a mixture of text and commands; the commands give detailed instructions to the formatting program about how the text is to be laid out on the page. These commands are normally low-level requirements in the domain of the finished document: "indent this margin by 1cm" or "change to font 3". Sometimes programs use the "compiler" paradigm: text and commands are placed together in a file which is then compiled to produce the formatted output. Alternatively, as in BRAVO [5], the "editor" paradigm is used: the formatting commands are not stored in the file that is under edit, but are executed immediately as they are entered.

Each paradigm has its strengths and weaknesses. Compiler-model systems have the problem that the user has difficulty visualizing the effect that his commands will have on the format, especially if he is unfamiliar with either the formatting program or the printing device. Interactive formatters do not preserve the original commands that yielded the format, so that if reformatting is necessary, no systematic change to the commands themselves is possible. In both cases, the onus of typographic quality and consistency falls on the user; he must be careful to be consistent in his use of margins, spacing, fonts, and the like, and he must avoid using combinations that will yield unattractive or cumbersome results.

After several years of use and observation of various document preparation software, we decided to create a new system, called Scribe, embodying ideas that had been generated from experience with these systems. Although the thought of an interactive formatter was intriguing, we retained the compiler paradigm partly because it would be easier to judge the success or failure of the finished product

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1980 ACM 0-89791-011-7/80/0100-0024 $00.75

in comparison with existing programs, and partly because we had no access to interactive terminals that were comparable in quality to our printing devices.

3. Design Strategy

In designing a better document formatting system, it behooves us to look at the best such system (a good secretary) to learn what we can and mimic what we can. The author provides text to the secretary, who visually parses the text into its components and makes some decisions about the nature and required format of the document. The secretary then selects an appropriate format and performs the necessary typing. Note well what is *not* communicated from author to secretary: any information about indentation, margins, spacing, and so on. All such typographic details are provided by the secretary, and in fact they differ from one format to the next even if the text is the same: a business letter and an internal memorandum typed from the same dictated text would have greatly differing appearances.

Since our requirements suggest that the author should not be permitted to specify low-level formatting detail, and since we know of at least one successful document formatting system (our secretary) in which the author does not specify such details, we choose to restrict the specification language so that the author *cannot* specify such details. But those details are necessary to produce the final document, so some equivalent must be provided. Our goal is to put the expertise in the program, in much the same style as Shortliffe's MYCIN puts bacteriological expertise into the program [9]. The formatting program will have access to a data base of formatting knowledge that it uses to achieve quality typography.

We shall require that each document produced by Scribe be of a specific declared *document type*, for example, a business letter or a technical report or a journal article. Each document type will have variant forms: a CMU letterhead, a journal article in ACM proceedings format. Scribe retrieves from its database the definitions appropriate to produce that document type on the intended printing device. Document type definitions are created by a format designer, a person with typographic expertise and knowledge of the format requirements of different documents. They are not usually created by an author.

4. Design of a New Document Formatter

We enumerated a set of requirements, some made of the system as a whole and some made of the specification language, then formulated a design to try to meet those requirements.

4.1. System Requirements

The design of the Scribe formatting language and its compiler was shaped by a number of specific requirements for the character of the resulting system.

In this and following sections, we use the term *manuscript file* to describe text prepared by the user for input to the compiler, and *finished document* for the result produced by compiling that manuscript file.

DEVICE PORTABILITY. Manuscript files should be independent of any particular printing device; the document specification language should therefore contain no constructs that will permit a user to put device dependencies in his file. Since the format of the document must vary from one printing device to another, this implies that the compiler must be able to supply the details.

SITE PORTABILITY. The manuscript form of a document should contain nothing that implicitly restricts it to a particular installation, which is to say that explicit file names and directory names and other local system names should not appear in the manuscript file. Where the manuscript must refer to the file system, it should be through indirect symbolic names and not direct file names.

SEPARATION OF CONTENT AND FORMAT. The text should contain no explicit formatting instructions, in order that it may be re-used in a different format or context. Since the explicit formatting must occur before paper output can be produced, the compiler must be able to supply the details. A corollary to this requirement is that the author should supply only the text, and a person

more knowledgeable in typographic matters should provide the typographic detail. Obviously a typographer cannot edit every manuscript file or provide the information explicitly each time, which means that there must be a mechanism for storing, retrieving, and using typographic knowledge.

INCREMENTAL CHANGE. Given that the format of a document is determined not by commands placed in the manuscript file but by the contents of a database, it should be possible to specify systematic changes to that format by placing suitable modification commands at the beginning of a manuscript file. Those modification commands must be incremental: the user need specify only those properties that he wants to change, rather than respecifying the entire definition.

We will restrict all modification commands to the beginning of a manuscript file to make it easier to remove them for purposes of re-use and portability.

RESTRICTED GOALS. Formatting problems whose solution is too difficult or whose specification is not possible in the specification language will be declared to be "beyond the scope of the system", and no attempt will be made to provide loopholes for them. We want to maintain the conceptual purity of the system at the expense of completeness, to make it easier to evaluate the finished product.

4.2. Specification Language Requirements

Rather than viewing the non-text parts of a manuscript file as commands to the formatting program, we view them as statements in a high-level non-procedural document specification language [12]. The author indicates the formatting effect desired, strictly non-procedurally, and leaves to the compiler the details of how to achieve that effect within the constraints of the selected printing device and document type. Certain requirements were placed on the language design.

DISCRIMINATION OF TEXT FROM SPECIFICATION. The major syntactic requirement of this specification language is that it not be confused with the text or interfere with it in any way. There should never be any ambiguity as to whether a particular lexeme is text. If multiple colors were available, we could mark the commands in colors, much as a proofreader does when proofreading a galley. If multiple fonts were available we could place all commands in some special font, such as script.

REPRESENTATION IN PRINTING ASCII. We intend to use the manuscript form of the document as the communications form which is transmitted to another site. We therefore require that a manuscript file, with text and specifications present in it, be representable using only the 95 printing Ascii characters.

NOT A PROGRAMMING LANGUAGE. The language in which an author communicates his needs about the format of his document is not to be a programming language. Procedural thinking does not come easily to non-technical people, and the system should be able to handle most of the formatting requirements of an undemanding user with simple nonprocedural specifications.

5. The Document Specification Language (Command Language)

A Scribe *manuscript file* is an unstructured stream of Ascii characters. It contains an interleaved mixture of text and statements that specify desired properties of the final format of that text. Although these statements in the specification language are not *commands* in the ordinary sense of the word (they do not directly command anything), we shall nevertheless call them commands as a matter of convenience. The "@" character was chosen as the sentinel to appear in text to flag the beginnings of commands: It is distinctive and is rarely used in normal prose, mathematical text, or program text. Each instance of an @ character denotes the beginning of a command. To get an @ character in text, two are used in the manuscript file: @@.

Every command has one of these forms:

@>	*a single character*
@Index(text)	*name and text*
@Set(Page=0)	*list of keywords and values*

If the *text* or *list of keywords and values* is empty, it may be elided along with its delimiters, provided that there is no ambiguity:

@*name*

Although parentheses are used in the description of the commands, any of Scribe's delimiter pairs, described below, can be used. Some sample commands:

```
@i(word)  @Heading[Table of Contents]
@Section<The Compiler>

@:  @\  @!

@Style{References CACM, Footnotes "*",
       Doublesided}
@Newpage
```

Each command has a name, and most apply to a specific delimited piece of text. The text may be delimited in a begin/end form:

```
@begin(quotation)
A @i[non-procedural] program is a
prescription ... some machine organization.
@end(quotation)
```

or in an abbreviated form:

```
@Quotation[A @i[non-procedural] program
is a prescription ... some machine
organization.]
```

Any delimiter pair from the set (...), [...], <...>, {...}, "...", '...', and `...' may be used, and the delimiters may be nested freely.

Semantically, most statements in the specification language are labels marking text in specific *formatting environments*. Each environment places certain requirements on the appearance of its text in the final document, without giving specific details. For example, an "Italics" environment (@I) requests that its text be set in an italic font appropriate to the surrounding text, or be underlined if there is no such font. A "quotation" environment (@Quotation), suitable for displayed quotations, requests that its text be set in a manner befitting quoted material, which in most document designs means that the left and right margins are adjusted inwards and the line spacing is made closer together:

> A *non-procedural* program is a prescription for solving a problem without regard to the details of *how* it is solved. That is, solutions should be specified implicitly in terms of structures or abstractions which are relevant to the problem rather than those operations, data, and control structures that are convenient for some machine organization [6].

There are about a hundred scalar variables whose values together define the environmental state, and therefore the details of the resulting format. Each environment is a partial specification of the state vector. The "italics" environment doesn't affect the state cells that determine margins, and the "quotation" environment doesn't normally affect the type face in used, so that "italic" and "quotation" are orthogonal. "Italic" and "Roman", however, both specify different values for the same state variable.

When environments are nested, the unspecified properties are simply inherited from the surrounding environment. Some environments, such as "chapter heading", specify almost everything: a particular font, boldness, size, spacing, margin, and so forth. Others, such as "italics", specify almost nothing, leaving everything but the face code attribute to be inherited. It is perfectly meaningful to specify a theorem inside a footnote or a footnote inside a theorem, though it is up to the document type designer to ensure that both are typeset reasonably.

A few actual commands exist in the document specification language; they correspond roughly to declarations in a programming language. For example, the @Device command declares the printing device, and the @Pageheading command declares a page heading to be used from that point on.

6. The Compiler

A Scribe compiler has been written in Bliss for a PDP-10; it compiles manuscript files into finished documents. Each manuscript files contains a (possibly implicit) specification of a device type (line printer, phototypesetter, etc.) and a document type (letter, thesis, journal article, etc.). The compiler loads from its database the definition of the printing device and the definition of the document format, then processes the manuscript file accordingly.

The essence of the compiler is its environment-processing mechanism. Whenever an environment transition occurs—entry or exit—the state changes are examined to see if they imply some sort of non-trivial processing. If so, the processing is performed. For example, if an environment entry is found to contain a request to use a large font on the phototypesetter, then the appropriate "rotate lens

turret" code will be generated; when that environment is ended and the containing environment is resumed, another turret-rotation code will be generated to position it back again.

An environment definition can specify, for each cell in the state vector, an absolute value (set the left margin to one inch), a relative value (widen the left margin by three centimeters), or no value at all (don't touch the left margin). The environment is represented internally as a list of (variable, value) pairs, and at the lowest level the environment-entry code just iterates over that list, making the appropriate changes to a copy of the old state vector. The kind of change to be made is represented by the "type" of the value in the compiler. The modification facility required in section 4.1 is implemented by providing a mechanism to edit these lists.

Although the compiler is strictly one-pass, it provides a full cross reference capability by means of an *auxiliary file* that is generated as a byproduct of the compilation. The auxiliary file is essentially an extract of the symbol table. Each time a manuscript file is processed, the compiler checks for the presence of an auxiliary file left behind by the previous compilation. The auxiliary file contains all of the cross-reference label definitions from the previous compilation, and usually they haven't changed much. If the compiler discovers that a definition of a label has changed (and hence the value used in a cross reference was wrong), it will print a warning message; if the manuscript file is recompiled without change, the cross references will all be correct.

7. The Database

Each time the Scribe compiler processes a manuscript file, it retrieves from its data base the information defining the relevant device properties, text formats, fonts, numbering schemes, and so forth. The contents of each database entry is just a sequence of Scribe commands; they are reparsed and re-evaluated in every compiler run.

The *device* files in the database provide device parameters (paper size, font size, spacing resolution, etc.) and also provide device-dependent implementations of the standard basic environments. For example, the "italic" environment is implemented separately in each device file. The device file for the line printer implements "italic" by turning it into "underline"; the device file for a photocomposer implements "italic" by changing the font code.

The *document type* files in the database provide device-specific environment format definitions for various document types. Each document-type file is partitioned into segments; each segment defines that format for a particular printing device or group of devices. Environment definitions do not necessarily occur in all document types—there is no need for a *return address* environment in a thesis or a *table of contents* in a letter—but careful attention is paid to making the names of environments that are conceptually similar be the same from one document type to another.

Bibliography format files in the database define the bibliography formats that are required by various journals. The *font family* files define various font combinations that have been selected by graphics designers to look harmonious when used together. The *character width* files define individual width and kern values for specific fonts.

We intend that the database files be created and maintained by persons with a higher level of expertise than that required simply to use the system. The device definition files are assembled by people with a relatively sophisticated knowledge of the workings of the compiler, and the document type files are intended to be created by persons with artistic talent (although the initial document type designs were all created by the compiler writer).

8. Scribe as a Writer's Workbench

Although its ultimate job is to format a printed document, the Scribe system is strongly oriented towards providing support to writers while a document is under construction. The name *Writer's Workbench* is borrowed from the Programmer's Workbench [2]. Many of these writer's tools are made possible or made simpler because of the high-level nonprocedural nature of the specification language.

The Scribe compiler will automatically generate numbers for pages, chapters, footnotes, theorems, examples, or what have you. One of the attributes that an environment may specify is numbering style, which includes a specification of how to number, how and when to increment the number, and how to display the number in the user's text. The cross-referencing mechanism allows references to be made to generated numbers by binding symbolic names to them, and all cross references are specified in terms of those symbolic identifier names. The compiler will print out charts, in sort by manuscript position and also alphabetically by identifier name, of the location of the symbolic labels, their values, and the references made to them.

Portions of a document that are essentially derivative—index, table of contents, glossary, table of figures—can be generated automatically. They are generated as manuscript files and then read back in as if they were part of the original document.

The document may be divided into a tree structure of subfiles; when a compilation is begun from the root the complete document is produced. Because of the existence of the auxiliary file (see section 6), it is possible to compile any subtree or leaf file separately and have all of the numbering and cross referencing come out right. When a non-root compilation is begun the compiler refers to the auxiliary file, in which it has saved enough information to enable it to restore its state to what it would have been had the compiler reached the same spot[1]. At the end of a non-root compilation, the auxiliary file is updated as necessary.

9. Experience and Conclusions

The Scribe compiler has been operational at Carnegie-Mellon since February 1978; manuals were available by summer 1978 [8], and the system was fairly stable by early fall. It is now at use at a number of other installations. As a production software system, it has been an unqualified success, which to a certain extent obscures our ability to learn about the success or failure of the ideas that were the essence of the research.

9.1. Successes

The specification language has turned out to be genuinely site-portable. It is commonplace to move files from one Scribe site to another and recompile them without incident.

The goal of device independence has been met well enough that people regularly make proof copies of a document on one device and the final copy on another. People who have never used any device but a line printer and never seen a photocomposer can change the @Device command in their manuscript files and produce handsome typeset output.

The language is simple enough that people with no programming background can learn it in a few hours of instruction, yet powerful enough that an 800-page textbook with fairly stringent formatting requirements has been typeset with Scribe [13]. A freshman class in "computer tools for humanities majors'" is now taught at Carnegie-Mellon; the students learn Scribe in the first few weeks of class and use it for class assignments.

The goal of separation of content and format was met well enough that we were able to produce this year's edition of our annual *Research Review* by merging the five papers as formatted by the authors with various tabular material, changing the document type, and producing a typeset copy. None of the papers contained any commands or environments that required any change before the five papers could be merged into a single journal.

9.2. Failures

The quality of typography, especially page layout, figure placement and widow elimination[2], is not good. This could be improved by writing a better compiler; there seems to be nothing intrinsic to the

[1]It is worth noting that the representation of environments as attribute/value pairs rather than as macros makes this process fairly simple.

[2]In publishing parlance, a *widow* is the last or first line of a paragraph that falls by itself at the beginning or end of a page.

specification language that limits typographic quality. Knuth's TEX has demonstrated that superb-quality typography can be obtained from computer-origin text [4]. We would like not to require as high a ratio of commands to text as does TEX.

The original specification language design did not include any facility for tabular material or mathematics, and it has proved difficult to add them gracefully after the fact, because it would require incompatible changes to the existing syntax to accommodate the extended specification language. It is not clear whether or not the essential notions of abstract non-procedural specification can be extended to mathematical text; it is quite certain that they can be extended to tabular material. The EQN and TBL preprocessor systems for TROFF [3, 7] demonstrate the feasibility of such a scheme, but neither system is entirely portable or complete.

The separation of content and format has made it difficult for users to circumvent errors or weaknesses in the compiler. The compiler needs to be fairly well debugged (which it often wasn't) for it to be usable at all.

The flexibility of the system, while a blessing for those who understand it, made it a nightmare to document. Almost no statement about the behavior of the compiler or the semantics of an environment can be universally true, because the definition of almost anything can be changed. This sort of flexibility needs to be imbedded in an interactive program having an extensive help facility in order that a user can find out not what an environment might do, but what it will do. The flexibility morass has the characteristic that novice users and expert users have no trouble; the intermediate-level users, who are knowledgeable enough to know how to change a definition, often don't understand the full nature of the change they are making and become very confused by the results.

The compiler is relatively slow; it processes only about 50 to 80 output pages per minute of CPU time on a DEC KL-10. Although this is an order of magnitude slower than simpler conventional programs like RUNOFF, the total amount of CPU time expended during the course of developing a document is often less, because the file need not be compiled as often.

9.3. Future Work

We are certain that the overall approach that shaped the design of the Scribe system is sound, but the compiler and database have grown to be fairly complex. Future efforts should be more interactive, should be more closely tied to the editor and file system, and should provide a more complete specification language *before* acquiring a user community.

With proper cooperation of file system, editor, and display device, and with careful design of the manuscript file specification language, it should be possible to design an interactive system that can maintain a BRAVO-like graphic display of the finished document while actually storing the commands that were used to generate that display.

Acknowledgements

Bill Wulf, Mary Shaw, and David Lamb inspired the Scribe project and performed the studies that made us believe it was feasible. Paul Hilfinger has managed to maintain the conceptual purity of the specification language in the face of hundreds of demanding users. Ivor Durham was the pilot user of the compiler, and Craig Everhart, David Lamb, Bruce Leverett, Paul Knueven, Philip Lehman, James Adams, Loretta Guarino, Elaine Rich, and Mike Kazar all helped the implementation in various ways. Larry Tesler's PUB provided a sandbox in which to test many ideas before beginning an implementation. Janet Walker made the User's Manual readable and finite.

A special thanks to the whole computer science community at CMU, who lived with a changing system for two years as I continually redefined it out from under them.

References

1. Gorlick, M.; Manis, V.; Rushworth, T.; van den Bosch, P. and Venema, T. *Texture User's Manual.* Department of Computer Science, University of British Columbia, Vancouver, B.C. V6T 1W5, 1975.

2. Evan L. Ivie. "The Programmer's Workbench - A Machine for Software Development." *Comm. ACM 20*, 10 (October 1977).

3. Kernighan, Brian W. and Cherry, Lorinda L. "A System for Typesetting Mathematics." *Comm. ACM 18*, 3 (March 1975), 182-193.

4. Knuth, Donald E. TEX: A System for Technical Text. Tech. Rept. AIM-217, Stanford University, November, 1978.

5. Lampson, Butler. *Bravo Manual.* Xerox Corporation, Palo Alto, CA, 1978.

6. Leavenworth, Burt M. and Sammet, Jean E. An Overview of Nonprocedural Languages. Proceedings of a Symposium on Very High Level Languages, ACM-SIGPLAN, April, 1974. Published as Volume 9, Number 4, of *SIGPLAN Notices*

7. Ossanna, J. F. TROFF User's Manual. Tech. Rept. 54, Bell Laboratories, 1977.

8. Reid, Brian K. *Scribe Users' Manual.* CMU Computer Science Department, 1978.

9. Shortliffe, Edward Hance. *Computer-Based Medical Consultations: MYCIN.* American Elsevier, 1976.

10. Sibbald, Kern E. DPS User's Guide. Tech. Rept. CN-16.0, University of Maryland, April, 1976.

11. Tesler, Larry. PUB: The Document Compiler. Tech. Rept. ON-70, Stanford University Artifical Intelligence Project, September, 1972.

12. Winograd, T. "Beyond Programming Languages." *Comm. ACM 22*, 7 (July 1979), 391ff.

13. W. Wulf, M. Shaw, P. Hilfinger, and L. Flon. *Fundamental Structures of Computer Science.* Addison-Wesley, 1980.